A STUDY OF THE EFFICIENCIES IN THE MOBILE PROGRAMMING SYSTEM

by

Ernest Henry Henninger

United States
Naval Postgraduate School

THESIS

A STUDY OF THE EFFICIENCIES IN THE MOBILE PROGRAMMING SYSTEM

by

Ernest Henry Henninger

June 1969

A Study of the Efficiencies in the Mobile Programming System

by

Ernest Henry Henninger
Lieutenant (junior grade), United States Navy
B.A., University of Colorado, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1969

ABSTRACT

The Mobile Programming System was developed to provide the capability of moving programs from one computing machine to another with a minimum of difficulty. This paper is an initial study of the efficiencies involved in the development of a processor for a programming language via the system. To this end, a language processor was implemented through the system on a particular machine (IBM 360 Mod 67), and comparisons made with the same language processor implemented directly on the same machine. Although the results of this paper are taken from this specific case, they give an indication of the relative efficiencies that could be expected from other processors implemented in a similar way. A significant side benefit of the study is a simplified implementation process for the SNOBOL4 programming language.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

The author is indebted to G. A. Kildall for the guidance he provided as advisor. In addition, G. L. Barksdale, Jr. and especially E. A. Singer are acknowledged for the help they provided in completing this paper. The author wishes to thank R. E. Griswold of Bell Laboratories for the information he furnished. Special thanks are due to Professor W. M. Waite for all the direct and indirect help he provided to make this paper possible.

I. INTRODUCTION

The continuing development of new computers has resulted in considerable duplication of effort in the area of programming. This has been especially true with respect to the implementation of programming languages. As a result, considerable attention has been focused on the problem of transferring programs from one computing machine to another. The Mobile Programming System [1] is one of the systems which has been developed to provide a solution to this problem. This paper was undertaken as a preliminary study of the efficiencies involved in implementing a programming language through this system.

A. TERMS AND DEFINITIONS

The following terms and definitions will clarify the terminology used throughout this paper:

Assembly language. Assembly language is usually considered to be symbolic notation for the basic machine language of a computer. Although there is a very real difference between assembly and machine language, in the context of programming systems it is often convenient to consider assembly language as machine language.

Macro. A good definition of a macro is given by[1]:

> The term "macro" was first used to denote a feature of certain assembly languages which allowed a programmer to refer to a group of instructions as though they were a single instruction. By mentioning the name of the "macro-instruction", the programmer caused all of the component instructions to be inserted at that point in his coding.

---

[1]Waite, W. M., The STAGE2 Macro Processor, Department of Electrical Engineering and Graduate School Computing Center, University of Colorado, pp. 1-3, 1 June 1968

...a classical ... macro definition has the form:

MACRO NAME (P_1, P_2,...,P_n)

Code Body

END

The words "MACRO" and "END" serve to delimit the definition, "NAME" is the macro name, and "P_1" through "P_n" are formal parameters. The code body is a series of lines which may contain instances of the formal parameters.

The Macro definition can best be illustrated by an example. Consider the following macro definition named "ADD" defined in terms of FORTRAN:

```
MACRO ADD(P1,P2,P3)
      P1 = P2 + P3
END
```

A call on this macro has the form "ADD(A1,A2,A3)". This macro call is replaced by the code body with the actual parameters A1,A2, and A3 substituted in place of the formal parameters P1,P2, and P3. For example, if the statement"ADD(X,Y,Z)" appears in a program, it is recognized as an instance of the "ADD"macro and is replaced by "X = Y + Z". Although macros have many different forms and capabilities, the examples given here cover the basic macro concept.

Macro Processor. A macro processor recognizes occurrences of macro calls and performs the corresponding replacement of macro calls by the expanded code. The type and power of macros that can be defined for a specific macro processor depend largely on the form which the macro name and formal parameters are allowed to take.

Macro-assembly language. A macro-assembly language is a fixed set of statements acceptable as input to a macro processor, where each of the statements of the macro-assembly language has a corresponding (and arbitrary) macro definition. Note that a macro-assembly language is a set of macro calls which are expanded through the macro processor and the macro definitions to some arbitrary language. It is important to note that the macro definitions may be altered without changing the macro-assembly language. A macro-assembly language can be thought of as a machine-independent language developed through a two-step process:

1. Design of an abstract computing machine.

2. Specification of an assembly-like language in macro form to operate on the data units of this machine.

A language developed in this manner will be referred to as a macro-assembly language.

A well-designed macro-assembly language can be easily implemented on a variety of computers. One should be able to obtain an implementation on a particular machine by coding each of the statements in the macro-assembly language as a macro containing instructions in the actual assembly language of the machine.

Translator or compiler writing system. A translator writing system is any programming system which automates part of the process of implementing a compiler, interpreter, or other language processor.

II. THE MOBILE PROGRAMMING SYSTEM

A number of papers have appeared concerning translator writing systems [2], but few of the systems developed can be readily moved from one machine to another. The MPS (Mobile Programming System), however, is a translator writing system designed to allow the transfer of programs from one computing machine to another. The MPS satisfies two basic requirements:

1. It allows for the transfer of existing language processors from one machine to another with a minimum of difficulty.

2. It is flexible enough to allow changes in language definition, and extensions to programming languages and their implementations.

The system is based on two processors: SIMCMP and STATE2. SIMCMP is a very simple macro processor capable of translating simple substitution macros with single character parameters. STAGE2 is a more sophisticated macro processor which accepts macro definitions with arbitrary strings of characters as parameters. It is coded in an abstract machine language known as FLUB (First Language Under Bootstrap), and is capable of being compiled by SIMCMP. Since the SIMCMP algorithm is defined in FORTRAN and can be easily encoded in assembly language on most computers, it provides a rapid means of implementation for STAGE2 via bootstrapping.

Higher levels (system compilers, processors, etc.,) of the MPS are written in terms of macros which STAGE2 can translate. The system, then, may be visualized to be evolved in three levels as shown in Fig. 1.

There are basically two ways a programming language can be implemented. The first is to treat a statement in a programming language as a call on a macro coded in assembly language. If this approach is taken, STAGE2

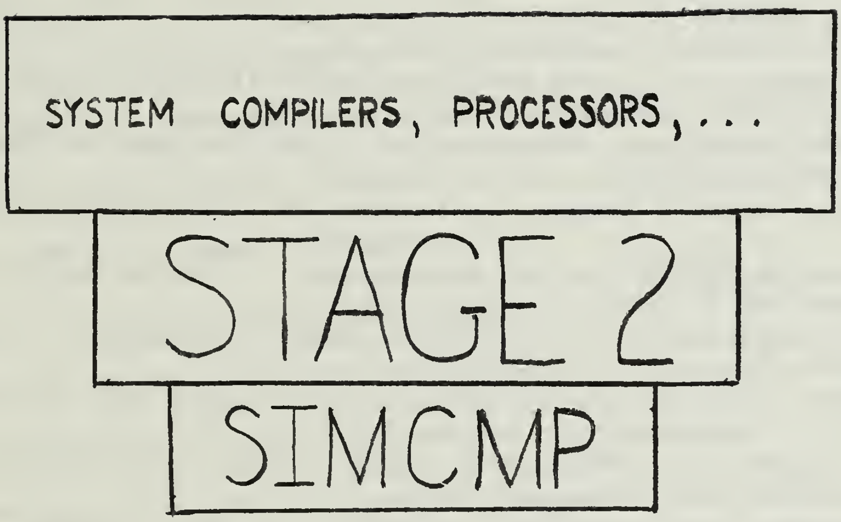

Figure 1

The Mobile Programming System

can be used to translate statements in the programming language directly to the assembly language of the desired machine. Although this has been successfully done for a portion of the BASIC programming language, it is an inadequate approach for programming languages of considerable complexity such as ALGOL and PL/1.

The second method of developing a compiler or other processor at the third level of the MPS is a two-step process:

1. The selection and specification of a machine-independent macro-assembly language suited to the coding of the processor.

2. The actual coding of the compiler or processor using the defined macro language.

This process is based on the observation that any assembly language coded compiler for a given problem-oriented language contains instances of certain operations characteristic of that language and the compiler (e.g., any assembly language implementation of FORTRAN must be capable of expressing FORTRAN boolean expressions). If these operations are properly extracted in terms of macro-assembly statements, a machine-independent compiler is obtained with the property that its implementation on a group of machines will not be too much less efficient than the same compiler hand implemented on each of the same machines. Here "efficient" is taken to mean the cost of development, maintenance, and operation on all present and future machines which the compiler is to be employed. The compatibility gained between machines for a particular language is a significant by-product of this process. As is generally the case, considerable effort is required in the initial development of a processor through this two-step process. When a compiler for a given language has been obtained, however, maintenance consists only of improvements to the compiler and changes due to extensions or modifications of the language.

Once a processor P written in a language L has been developed via the above process, movement from the original machine to a new machine requires only three steps:

1. First the SIMCMP macro processor is implemented on the new machine. SIMCMP is currently coded in FORTRAN, but may be easily encoded in another language on the machine. The MPS user defines macros in SIMCMP which allow expansion of simple macro-assembly language statements. FLUB is one such macro-assembly language which can be processed by SIMCMP. It is important to note that FLUB is the language, and SIMCMP is the processor.

2. Using the FLUB implementation obtained in step (1), the MPS user submits the STAGE2 macro processor to SIMCMP for compilation. STAGE2 is a macro processor written in the macro-assembly language FLUB. The principle advantage here is that the user has now implemented a very powerful macro processor (STAGE2) by first implementing an extremely simple macro processor (SIMCMP).

3. Recall that the language processor P has been encoded previously in the macro-assembly language L. For each statement of L the user defines a macro expansion for that statement. The macro expansion is in terms of the assembly language of the new machine. If there is no assembly language available on the new machine, the user merely translates to an available language (e.g., B5500 EXTENDED ALGOL on the Burroughs B5500[1]). One may then submit P (written in L) to STAGE2 for processing. The resulting assembly language is then compiled on the new machine, and an implementation of P is obtained.

---

[1]This is not an inconsistency in the use of the word "assembler" since one may easily consider the ALGOL compiler on the B5500 an assembler, albeit a very fancy assembler.

Note that SIMCMP is only needed to implement STAGE2. Also, the old machine is not involved in the process of implementing P on the new machine. If desired, however, the old machine could be used to eliminate steps (1) and (2), assuming an implementation of STAGE2 is available on the old machine. In general, little would be gained by this, as it is usually easier to work on the new machine than try to communicate between the new and old machine. Note also that the efficiency of operation of a final implementation of P at a particular installation will depend on how highly tuned it is to the installation. That is, the efficiency of P depends upon the degree to which the features of the new machine have been used in coding P in L and in the translation done by STAGE2. The level of tunning achieved will generally depend on the number of optimization features (e.g., conditional assembly) built into the L macros.

In general, a processor will be coded in a macro-assembly language of considerable power since this simplifies the coding. In addition, a processor's operation can usually be most accurately specified in terms of the macro-assembly language containing the greatest number of operations. Therefore, the efficiency of the final implementation of a processor should increase with the number of different operations in the selected macro-assembly language. Note, however, that the time required to transfer a given processor from one machine to another will also increase with the power and number of macro operations since each must inevitably be implemented. Thus, mobility and final efficiency are opposing factors.

If macro capabilities are available with the assembler for the new machine, it might be suggested that steps (1) and (2) above be eliminated and the L macro statements coded directly in terms of macros for the new machine. This approach could be taken, but STAGE2 has several features which make its use particularly suited to the task:

1. A generalized matching algorithm is used which allows for greater control over the matching process and resulting expansion of code.

2. STAGE2 is independent of any particular assembly language and operates solely as a code processor. This has two advantages:

   a. The user may work in any language or group of languages.

   b. This allows segmentation in the translation process, i.e., only part of a program may be translated if desired, or a given program may be processed in a sequence of steps with different macro definitions used at each step.

These features, combined with its simplified implementation process, make STAGE2 a useful base for the implementation of higher level processors. For a more detailed description of the operations and implementations of SIMCMP and STAGE2 see [1].

III. A PROPOSED APPROACH TO THE PROBLEM OF MAINTAINING MOBILITY

Assume that a group of processors $P_1, P_2, \ldots, P_n$ has been implemented through the MPS in macro-assembly languages $L_1, L_2, \ldots, L_k$. The most time-consuming part of the three-step process (refer to Section II), required to move these processors $P_1, P_2, \ldots, P_n$ to a new machine is step (3): the coding and debugging of the macros defined in STAGE2 which translate each of the macro-assembly languages $L_1, L_2, \ldots L_k$ to assembly language. This process is simplified for FLUB-like languages (any macro-assembly language which is an extension of FLUB) since FLUB consists of only twenty-seven operations, and a thorough test program is available to test these operations. If similar test programs are developed for each of the macro-assembly languages $L_1, L_2, \ldots, L_k$, the problem of debugging is minimized. However, the length of such test programs increases directly with the number of operations in each macro-assembly language. Therefore, this individual testing approach may prove to be impractical for those macro-assembly languages among $L_1, L_2, \ldots, L_k$ of considerable complexity. In addition, the existence of test programs would have no effect on simplifying the initial coding required in step (3). Thus it can be seen that although implementation of the processors $P_1, P_2, \ldots, P_n$ will certainly be much faster than the hand coding of each of them on the new machine, the time and effort required may still be considerable.

An alternate approach may be taken to the problem of maintaining a high level of mobility which tends to minimize the coding and debugging required to implement $L_1, L_2, \ldots, L_k$. Recall that the macros in step (3) need not expand the macro-assembly languages $L_1, L_2, \ldots, L_k$ to assembly language. Thus if each of these macro-assembly languages is first implemented in terms of a simpler macro language (such as FLUB), implementation of the processors $P_1, P_2, \ldots, P_n$ reduces to implementation of the simpler languages. Assuming that test programs are available for the simpler languages, the implementation process proves to be a relatively easy task.

Since FLUB is one of the few simpler languages for which a thorough test program is available, it is advantageous to make as much use of it as possible. Therefore, in apply the above bootstrapping technique to the development of a processor P to be coded in a macro assembly language L, L should initially be specified so that the macros can be expanded to a FLUB-like langauge. If this technique places too high a restriction on the form of L, or if the implementation of L obtained through FLUB-like operations proved to be excessively inefficient, another abstract language may be used which is better suited to the task. The important point to note is that a two-level technique is used to obtain a simpler implementation of P without sacrificing machine independence. Once P has been bootstrapped in this way it can be implemented on a particular machine by simply implementing the lower level language. Since the degree of mobility for a group of processors depends on the number of simpler languages and operations contained therein, it is desirable to keep both to a minimum.

Although a greater level of mobility is achieved for a group of processors implemented with the two-level process, their ultimate implementations on a particular machine will be less efficient since two levels of machine independence are involved rather than one. This is not as great a problem as it initially appears, however. In the installation of this group of processors on a particular machine it is likely that some will be used a great deal more than others (e.g., a list processing language such as LISP would probably be used much less than an algebraic language like FORTRAN). For those processors that are used infrequently it is not necessary to have a highly efficient implementation. Thus for this group of processors, the initial implementations obtained will be acceptable in most cases. For those processors which must be highly efficient, the implementation in terms of the lower level language will provide a rapid initial implementation. As in the case when only one macro-assembly language is used (refer to Section II), most of the inefficiencies introduced by the lower-level abstract machine language can be tuned out. In fact, since the higher level macro-assembly language could be implemented directly in terms of assembly-language coded macros, the implementation available in terms of the lower level language could be gradually changed to conform to a direct implementation. This can be done by taking the macro implementation available in terms of the simpler abstract language and replacing small groups of these macros by their direct implementations for the particular machine. The process continues in a series of steps until a debugged final implementation is obtained. Note that this multi-phase process provides a systematic approach to obtaining a debugged version of the direct implementation since errors introduced to the implementation obtained at each step are due entirely to the macros replaced at that step. In most cases the tuning process will not require recoding of each of the higher level macros implemented in terms of the lower level abstract language, but only those that have gross inefficiency when compared to a direct implementation.

Thus, it can be seen that in the long run nothing is lost by the use of the two level implementation process while three important advantages are gained:

1. Increased mobility for a group of processors.

2. It provides a systematic approach to the process of debugging.

3. It allows the implementor to concentrate his tuning efforts only on those processors for which highly efficient implementations are necessary.

Several questions arise with respect to the above two level implementation. First, if an implementation of a processor in a lower level language will increase its mobility, why not code the processor initially in terms of a simpler language? Although an implementation obtained in this way will certainly be more efficient than one obtained through two levels, the time required to code a processor makes it a long and difficult task. In addition, as previously noted, a processor can usually be better specified in terms of a language of considerable power.

Another point that may be questioned is that of even attempting to specify the higher level language in terms of FLUB-like operations. This approach appears backwards, i.e., it seems more reasonable that the higher level language should be specified to be suited to the coding of the processor, later extracting the lower level language. Actually, although FLUB was designed specifically for coding STAGE2, it was derived in exactly this manner. The data units on which the FLUB language operates were designed as an adaptation of those used in the implementation of SNOBOL4 [3], a string manipulation programming language implemented in a machine-independent macro-assembly language. Thus most of the FLUB operations are basic to the language in which the SNOBOL4 compiler is coded.

Since SNOBOL4 contains the characteristic qualities of a string manipulation language, FLUB reflects some of these qualities. Noting that string manipulation is common to most compilers or other language processors one might expect that FLUB would prove useful for implementating other processors if a FLUB-like language produces a reasonably efficient lower level implementation of the SNOBOL4 macro-assembly language. A natural first step in testing these ideas, then is to implement SNOBOL4 in terms of FLUB-like operations and compare this implementation with one obtained directly on a particular machine. This is the primary undertaking of this paper. Before giving a detailed description of the study, however, some insight may be gained by an examination of related work in this area.

IV. RELATED RESEARCH IN MOBILE PROGRAMMING SYSTEMS

One of the earlier attempts to specify a mobile programming system was that made by the Share Ad-Hoc Committee on Universal Languages [4]. They proposed a three level approach to the implementation of a programming language: a highest level composed of all current and future problem-oriented languages; a middle level consisting of a single language known as UNCOL, the Universal Computer Oriented Language; and a lowest level to be made up of all current and future machine languages (allowing symbolic machine-like languages such as assembly language). UNCOL was to be a universal language capable of expressing any computable problem, having many things in common with each of the lower level machine languages. Corresponding to each of the programming languages at the highest level there was to be a compiler written entirely in UNCOL. Thus to implement a programming language on a particular machine, it would only be necessary to write a processor to translate UNCOL to machine or assembly language for this machine. The primary problem with this approach is the difficulty in specifying UNCOL. The language would have to incorporate many of the features of a large group of machines and be capable of producing reasonable efficient implementations of all problem-oriented languages.

It is interesting, however, to note some of the similarities of the UNCOL approach to implementing a programming language and the three level method (refer to Section II), which may be used in the MPS. Both systems attempt to provide a rapid means of obtaining a reasonably efficient implementation of a programming language at the machine or assembly language level. In order that any system provide this capability it is necessary that it contain a measure of machine independence and mobility. In the UNCOL system both of these are provided by UNCOL. The mobility of the system depends on the effort required to implement an UNCOL to machine or assembly language translator. In the three level MPS method, machine independence is provided in both the macro-assembly language selected to code the language compiler and in the translator (STAGE2). Mobility in this case depends on the effort required to implement STAGE2 and the selected macro-assembly language in terms of STAGE2 macros. Note that in both cases the compiler for the programming language is implemented in terms of an intermediate machine-independent language, and an implementation is obtained by translation to the machine or assembly language level. Thus one might compare the UNCOL level in the UNCOL system to the union of all macro-assembly languages suited to the coding of a compiler for a programming language in the three level MPS approach.

A recent advocate of the macro approach to implementing a higher level language is Halpern [5,6]. He claims that a language is best described and implemented by a body of macro definitions defined directly in terms of machine language. To achieve this he suggests the use of a generalized macro processor, capable of allowing for the definition of new operators. The XPOP system was constructed as a prototype to test these ideas, and at least one language , ALTEXT [7], has been successfully implemented in the system. The primary problem with the XPOP system is the treatment of a statement in a programming language as a call on a macro coded in terms of assembly language or in terms of other macros coded in assembly language. As was previously noted, this may be reasonable for simpler languages (such as ALTEXT), but for more sophisticated languages, this approach is inadequate, even with a macro processor capable of allowing generalized operations. The role of STAGE2 in the MPS is comparable to the generalized processor proposed in Halpern's scheme.

One study which produced a reasonably rapid implementation of a programming language was made by Madnick [8]. He applied the boot-strapping technique technique to the development of a SNOBOL compiler specifically for the IBM 360 through a series of four steps[2]:

1. Specification of a basic string processing language (SPL/1) and implementation of an interpreter for the language.

2. Design and production of an assembler to produce input to the system.

3. Development of a SNOBOL compiler using the assembler and interpreter.

4. Linkage of the assembler-interpreter-compiler to the IBM 360.

SPL/1 was designed to enable easy expression of any SNOBOL Program. Thus to obtain an implementation of SNOBOL it was only necessary to develop a compiler capable of translating SNOBOL to SPL/1. This was done by coding the SNOBOL-SPL/1 compiler in SNOBOL, and then using this translator on a machine capable of running SNOBOL (IBM 7094) to translate the SNOBOL-SPL/1 compiler to SPL/1. This translation need only be applied once. The SNOBOL-SPL/1 compiler obtained from this process was then capable of running via the SPL/1 assembler-interpreter already coded in assembly language for the IBM 360.

---

[1]Cambridge Scientific Center, <u>SPL/1: A String Processing Language</u>, by S. E. Madnick, pp. 1-2, June 1966.

The SNOBOL compiler derived through this process proved to be an efficient interpreter of a program once compiled, but was only capable of compiling about 100 SNOBOL statements per minute. This inefficiency was primarily due to the compiler running interpretively. Another limitation of the SPL/1 system was the representation of symbol strings as linked single-word (4-byte) blocks, with one character stored per word. This only allows for the effective use of 25 per cent of the available storage, with the remainder used for pointer information. The restrictions imposed by compiler speed and use of storage necessarily limits the application of this system. Considering the entire system was developed in approximately three months, however, one can see that it represents a reasonably successful development of a mobile compiler. As was pointed out by the author, several improvements undoubtedly could be made to improve the running speed of the compiler. Although Madnick's work was associated entirely with the IBM 360, his techniques could be applied on other machines to produce a reasonably rapid (2-3 months) implementation of a SNOBOL-like compiler of similar or somewhat improved efficiency.

V. SNOBOL4 AND THE MPS

One programming language which is implemented in terms of macro-assembly language is SNOBOL4 [9,10]. SNOBOL is a string manipulation language developed at Bell Laboratories. The language has evolved over a period of years, and its implementation was put on a machine—independent footing in SNOBOL4. The implementation of the language is probably best described by one of its originators[1]:

> The SNOBOL4 programming language is implemented in macro-assembly language. This macro language is largely machine independent and is designed so that it can be implemented on a variety of computers. Thus, an implementation of the SNOBOL4 language can be obtained by implementing the much simpler macro language. By implementing the macro language, one obtains a version of the SNOBOL4 language which is largely source-language compatible with other versions implemented in the same way. Nearly all the logic of the SNOBOL4 language resides in the program written in macro language. Thus if one implements the macro language properly, the resulting implementation of SNOBOL4 will be essentially the same as other such implementations.

The SNOBOL4 language was not designed in the context of any particular programming system. It is generally assumed that the macro language will be implemented in terms of assembly language on a particular machine via any available macro processor (usually part of the assembler). Thus, SNOBOL4 can be implemented on any machine with an assembler and a macro processor capable of translating the SNOBOL4 compiler, written in macro-assembly language, into assembly language.

The SNOBOL4 language can be implemented via the MPS since it is coded in a machine-independent marco language. Using the MPS, SNOBOL4 can be implemented on any machine with an assembler.

[1]Griswold, R. E., A Guide to the Macro Implementation of SNOBOL4, Bell Telephone Laboratories, p. 3, 20 Feb 1969.

As was pointed out in the general discussion of the MPS, any programming language implemented in terms of a macro-assembly language would generally be coded in one of considerable power, and this is the case with SNOBOL4. The SNOBOL4 macro-assembly language consists of 129 operations, and although implementation of these macros is much simpler than hand coding of the compiler, the effort required is still significant. In addition, once these 129 operations are implemented and preliminary tests made on each, if the SNOBOL compiler fails to properly execute, debugging will likely prove to be a difficult task. The solution to these two shortcomings, proposed in Section III, was implementation of the macro-assembly language in terms of a simpler abstract machine language for which a test program was available. Therefore, SNOBOL4 was implemented in terms of a FLUB-like language as a first step toward studying the feasibility of the proposed two-level macro-assembly language approach to implementing a programming language.

Although the details of implementation will not be discussed in this paper, a few points should be made. SNOBOL4 programs execute interpretively. Thus in implementing SNOBOL4 in terms of FLUB-like operations, it was not necessary to face the problem of placing compiler code emitted on a machine-independent footing. If one were implementing a machine-independent version of FORTRAN, for example, it would be possible to code the compiler in a macro-assembly language emitting a macro-assembly language. This compiler could, if properly coded, be translated using STAGE2 as a translator to an assembly language compiler. The assembly language compiler then emits assembly language for the desired machine. The possibility of this type of implementation is an area for future study.

One undesirable characteristic of the SNOBOL4 implementation process is the fact that SNOBOL4 Input-Output corresponds to that of FORTRAN. Thus implementation of SNOBOL4 I/O is a difficulty unless an implementation of FORTRAN is available. Since the FLUB I/O operations are much easier to implement, this problem is minimized by coding FORTRAN-like I/O routines in terms of the FLUB I/O operations. Although this results in the loss of some I/O capabilities, it may be justified by a gain in mobility. In practice, if some higher level language I/O capabilities are available (e.g., FORTRAN), there is usually not too much mobility lost in taking advantage of them.

The results of the implementation will be discussed in the next section. It should be noted that all results and implementations discussed were obtained on an IBM 360 (Mod 67).

VI. RESULTS OF THE TWO-LEVEL IMPLEMENTATION OF SNOBOL4

There are primarily two areas of interest to be examined in the implementation of a programming language through a mobile system:

1. A measurement of the mobility of the system. This mobility is measured by the effort required in implementing the language.

2. The efficiency of the final product.

The SNOBOL4 compiler is a mobile program. The mobility and efficiency is examined briefly below.

As was previously noted, the SNOBOL4 macro-assembly language (hereafter referred to as S4) consists of 129 operations. Although the time required to implement these will vary from machine to machine, it can be roughly estimated that on a particular machine they can be implemented in 50-150 man hours or approximately 3-6 months by an individual. On the IBM 360, for example, the assembly language macro definitions consist of about 2000 statements. The associated subroutines also comprise about 2000 statements.

The efficiency of a macro-coded SNOBOL4 compiler implemented in assembly language as compared to a similar compiler hand implemented in assembly language is a much harder question to answer. The true efficiency can only be answered by comparison of the two implementations. Although this is a major question that needs to be answered with respect to the macro approach to implementing a compiler, it was not undertaken as part of this study.

Some idea of the execution times associated with a SNOBOL4 program can be taken from TABLE I. TABLE I provides a list of timings and memory requirements for the compilation and execution of four programs

TABLE I

TIMINGS AND STORAGE REQUIREMENTS
FOR SNOBOL4 (VERSION 2.0)

A. Timings for Four Sample Programs
(seconds CPU time)

	I	II	III	IV
(1)	2.40	11.29	5.51	2.52
(2)	2.35	10.03	4.76	2.17
(3)	2.1%	11.2%	13.6%	13.9%

B. Storage Requirements
(1000 bytes)

	M_1	M_2	M_3
(1)	104	176	191
(2)	92	164	179
(3)	11.5%	6.8%	

Note: (1) gives the values for the SNOBOL4 compiler obtained from direct expansion of the S4 source to assembly language.

(2) gives the values for the optimized version of the SNOBOL4 compiler used in (1).

(3) shows the percent improvement in (2) over (1), i.e., ((1)-(2))/(1).

M_1 gives the basic storage requirements for the main program and subroutines (values rounded to the nearest thousand bytes).

M_2 gives the minimum storage required for execution (includes IBM 360 operating environment).

M_3 shows the actual storage used to obtain the timings in (A).

run on two different implementations of the SNOBOL4 compiler. A description of each of these sample programs is included in APPENDIX A. The first set of timings in TABLE I are those for the version of SNOBOL4 obtained from direct macro expansion of the S4 source to assembly language. The second set of timings shows the results for the same set of sample programs run on the distributed version (2.0) of SNOBOL4. This is a somewhat optimized version of the compiler obtained from direct macro expansion. For the sample programs included, it can be seen that the optimized version runs approximately 2-15% faster. The actual spread between the two compilers, however, is probably closer to 2-25%.

The FLUB-like language (hereafter referred to as EFLUB) used to implement the SNOBOL4 compiler consists of 40 statements. The STAGE2 macro definitions which translate S4 to EFLUB consist of approximately 1500 statements and the associated subroutines comprise about 1000 statements in EFLUB.

It should be noted that of the 129 S4 operations appearing in the SNOBOL4 compiler, implementation of 23 of these is optional. Twenty of these optional operations were not implemented in the EFLUB implementation. The primary feature disabled by omission of these was a real arithmetic capability. When this feature is added to the implementation, approximately five new operations will be added to the existing 40, and the length of the macro definitions and subroutines should each increase about 5%.

The STAGE2 macro definitions which translate EFLUB to assembly language (IBM 360) consist of approximately 500 statements, about half of which correspond to the 4 I/O operations used. The steps required to obtain a running version of SNOBOL4 through the MPS are summarized below:

1. Hand coding SIMCMP in assembly language (2-5 man hours).

2. Implementation of FLUB (in terms of SIMCMP macros coded in assembly language) and translation of STAGE2 (via SIMCMP) to obtain its implementation (3-7 man hours).

3. Implementation of EFLUB (in terms of STAGE2 macro definitions coded in assembly language) and translation (via STAGE2) of the SNOBOL4 compiler to assembly language (4-10 man hours).

A reasonably competent programmer should be able to obtain an implementation of SNOBOL4 through the above three-step process in 1-3 weeks. The exact time, of course, depends upon computer accessibility. Note that step (3) assumes that the user starts with a SNOBOL4 compiler already translated from S4 to EFLUB. Improvements to the implementation obtained can be made by making assembly language modifications to the macros which translate S4 to EFLUB. Improved versions of SNOBOL4 can then be generated by using both these modified macros and the macros coded in step (3) to translate the S4 source to assembly language.

If a higher level programming language is already implemented on the machine being used, steps (1) and (2) can be simplified by coding SIMCMP and the FLUB macros in step (2) in terms of this language. This, of course, will result in a less efficient version of STAGE2. If FORTRAN is available, for example, steps (1) and (2) require very little work since SIMCMP is defined in FORTRAN, and the macros which translate EFLUB to FORTRAN are provided with the MPS.

It is interesting to consider the computing times involved in each of the above steps required to obtain a running version of SNOBOL4. A FORTRAN version of SIMCMP is capable of translating STAGE2 to assembly language or FORTRAN in approximately thirty seconds CPU time (IBM 360/67).

Although exact figures cannot be given for translating S4 to EFLUB, since translations were performed in segments, a FORTRAN version of STAGE2 is capable of translating S4 to EFLUB in approximately 35 ($\pm$ 7) minutes. An assembly language version should be able to perform the same translation in approximately 15 ($\pm$ 4) minutes. Translation of EFLUB to assembly language should require about the same corresponding time for each of the versions. It can be seen that in dealing with times of this magnitude, segmentation of the translation process can be a useful tool.

It should be noted that although the times associated with the above process need only be tolerated until an implementation of SNOBOL4 is obtained. Thus, the important question is not how costly it is to obtain an implementation, but how efficient the final implementation is. Final debugging of the EFLUB implementation of S4 is not yet complete. As a result, actual timings of an EFLUB SNOBOL4 compiler[1] are not yet available. However, it was possible to obtain a computed estimate of the final results. This was done by first computing the time required to execute each of the S4 macro operations implemented directly in assembly language and the times required to execute each of the corresponding operations implemented in terms of EFLUB and then translated to assembly language. All computations were made using available timing charts for the IBM 360 (Mod 67) instructions. The computed times for all S4 operations acting on strings were based on a string length of 25 characters in both cases. These computations provide an estimated ratio of the time required to execute an S4 operation implemented in EFLUB (T_i') versus the time required to implement the same operation

[1]Obtained by translating an implementation of S4 written in EFLUB to assembly language.

implemented directly in assembly language (T_i). By themselves, these values provide little information on the final efficiency of the EFLUB implementation. However, a table of "frequency of use" for each of the S4 operations is provided in [10]. These values were obtained from statistics runs on a large number of SNOBOL4 programs. Although the true values will vary from program to program, these can be considered representative of a typical SNOBOL4 program.

Based on the frequency of use values (F_i) and the ratio of execution times (T_i'/T_i) it is possible to compute the contribution of each of the S4 operations to the final efficiency by $(T_i'/T_i)F_i$. By summing these computations over the 109 S4 operations (20 optional operations were not implemented) one obtains the approximate ratio of execution time of an EFLUB implementation over an implementation from direct macro expansion of the S4 source to assembly language.

The results of the computations for the 109 operations were order by increasing (T_i'/T_i) values and separated into five groups:

1. Those operations with a ratio $0 < (T_i'/T_i) \leq 1.4$
2. Those operations with a ratio $1.4 < (T_i'/T_i) \leq 2.4$
3. The remaining operations
4. Those operations in (1) or (2)
5. All the operations

The cutoffs 1.4 and 2.4 are arbitrary and were selected because they separate the EFLUB implementations of the S4 operations into three groups of decreasing efficiency. TABLE II summarizes the results of the computations for these five categories. Note that column V gives the values obtained for a SNOBOL4 compiler implemented using the EFLUB implementations of the S4 operations in the group and hand coding

TABLE II

COMPUTED TIMINGS AND STORAGE REQUIREMENTS
FOR EFLUB SNOBOL4 COMPILER

A. Computed Timings

	I	II	III	IV	V
(1)	91	.55	.72	1.31	1.17
(2)	11	.25	.57	2.28	1.32
(3)	7	.20	1.25	6.25	2.05
(4)	102	.80	1.29	1.61	1.49
(5)	109	1.00	2.54	2.54	2.54

B. Storage Requirements
(1000 bytes)

	M_1	I_1	M_2	I_2
(6)	92		164	
(7)	104	13.0%	176	7.3%
(8)	138	32.7%	210	19.3%

Note:
(I) gives the number of S4 operations of the total 109 implemented in the EFLUB SNOBOL4 compiler included in each of the categories (1) through (5).

(II) gives the total frequency of use in a typical SNOBOL4 program for the S4 operations included in each category.

(III) gives the sum of the values $(T_i'/T_i)F_i$ for each of the S4 operations included in the category. T_i is the time required to execute an S4 operation implemented in assembly language. T_i' is the time required to execute the corresponding S4 operation implemented in EFLUB. F_i is the corresponding frequency of use for each of the S4 operations.

TABLE II (continued)

(IV) gives the ratio III/II.

(V) gives the ratio of execution times for an EFLUB SNOBOL4 compiler over the unoptimized assembly language SNOBOL4 compiler. This ratio is based on the assumption that the EFLUB compiler is obtained by implementing each of the S4 operations in the category in EFLUB and the remainder in assembly language.

(6) shows the values for the optimized S4-assembly language SNOBOL4 compiler.

(7) shows the values for the unoptimized S4-assembly language SNOBOL4 compiler.

(8) shows the values for the SNOBOL compiler implemented entirely in EFLUB.

M_1 gives the basic storage requirements for the main program and subroutines.

M_2 gives the minimum storage required for execution (includes the IBM 360 operating environment).

I_1 indicates the percent increase in M_1 storage requirements with respect to the previous entry. For example, under (8) I_1 gives ((8)-(7))/(7).

I_2 indicates the percent increase in M_2 storage requirements with respect to the previous entry.

(optimizing the internal coding) the remaining operations in assembly language. It can be seen that by taking the EFLUB implementations in group IV and simply hand coding seven macros one obtains a SNOBOL4 compiler with a final execution ratio of 1.49. The amount of hand coding is minimal (less than 200 statements in assembly language on the IBM 360).

TABLE II also gives the estimated storage requirements for the EFLUB SNOBOL4 compiler. Comparisons with the same values for the two S4-assembly language SNOBOL4 compilers found in TABLE I are included. It can be seen that the EFLUB version main program and subroutines require about 32.7% more storage than that required for the SNOBOL4 compiler obtained by direct expansion from S4 to assembly language. Considerable decrease in this value can be expected if some optimizing of the EFLUB source for the SNOBOL4 compiler is made.

Although it is difficult to estimate the error in the EFLUB timings and storage requirements, values for each should be accurate to within 15%.

VII. CONCLUSIONS

The results in TABLE II provide some estimate of the efficiency of the EFLUB implementation, and are based on the efficiencies that can be expected over a large number of programs. These figures do not, however, provide an upper or lower bound on the actual efficiencies involved. Thus, the first step toward completing the study is to verify and expand upon the estimates in TABLE II.

The computations associated with each of the S4 operations which comprise TABLE II suggest several improvements which can be made in the EFLUB implementation. In addition, no results are available on the efficiencies that can be expected from an optimized final version of an EFLUB SNOBOL4 compiler. A 2-25% increase in efficiency was gained by optimizing the version of the SNOBOL4 compiler obtained by direct macro expansion of S4 to assembly language. One would expect a similar gain in efficiency if the EFLUB version of the SNOBOL4 compiler were optimized. Thus, considerable improvement could be made in the final version of the EFLUB SNOBOL4 compiler.

Finally, two questions need to be answered with respect to the macro approach to implementing a compiler for a programming language:

1. How efficient is an implementation of a programming language obtained from a macro-coded compiler as compared to the implementation of the same compiler hand coded in assembly language?

2. How much can the implementation process be simplified for a macro-coded compiler without significantly effecting the efficiency associated with (1)?

The first question was not considered in this paper. The results of this paper suggest the following answer to the second question. If a machine-independent macro-coded compiler for a programming language is implemented first in terms of a lower level machine-independent macro language, the MPS and this lower level implementation can be used to obtain a 2-4 week implementation of the programming language. Further, with some optimization, this technique will not significantly effect the efficiency associated with the original compiler.

In the final analysis, the extent to which macro-coded compilers are developed will depend on the answer to the first question posed. The mobility associated with macros, however, has shown that they can be a useful tool in the direct or indirect implementation of programming languages.

APPENDIX A

DESCRIPTION OF SNOBOL4

SAMPLE PROGRAMS

SAMPLE PROGRAM I

DESCRIPTION: This program performs a topological sort. It maps a partial ordering of objects into a linear ordering A(1), A(2),..., A(N) such that if $A(S) < A(T)$ in the partial ordering, then $S < T$.
LENGTH: 47 statements.
INPUT: 15 relations.

SAMPLE PROGRAM II

DESCRIPTION: This program deals bridge hands using a random number generator. Three hands are dealt and printed.

LENGTH: 101 statements.

SAMPLE PROGRAM III

DESCRIPTION: This program computes and prints a table of N factorial for values of N from 1 through 45.

LENGTH: 30 statements.

SAMPLE PROGRAM IV

DESCRIPTION: This program computes the number of uses of each word in a string of text.

LENGTH: 14 statements.
INPUT: 71 word string containing 50 different words.

Note: Sample programs I, II, III can be found in [3].

REFERENCES

1. Waite, W. M., *The STAGE2 Macro Processor*, Department of Electrical Engineering and Graduate School Computing Center, University of Colorado, 1 June 1968.

2. Feldman, J., and Gries, D., "Translator Writing Systems," *Comm. ACM*, v. 11, pp. 77-113, February 1968.

3. Griswold, R. E., Poage, J. F., and Polonsky, I. P., *The SNOBOL4 Programming Language*, Bell Telephone Laboratories, Holmdel, New Jersey, 6 August 1968.

4. Share Ad-Hoc Committee on Universal Languages, "The Problem of Programming Communication with Changing Machines A Proposed Solution," *Comm. ACM*, v. 1, pp. 12-15, August 1958.

5. Halpern, M., "XPOP: A Metalanguage without Metaphysics," Proc. AFIPS 1964 FJCC, Vol. 26, pp. 57-68.

6. Halpern, M., "Toward a General Processor for Programming Languages," *Comm. ACM*, v. 11, pp. 15-26, January 1968.

7. Lockheed Missiles & Space Company Tech. Rep. 6-75-65-15, *ALTEXT- Multiple Purpose Language*, by R. Stark, March 1965.

8. Cambridge Scientific Center, *SPL/1: A String Processing Language*, by S. E. Madnick, June 1966.

9. Farber, D. J., Griswold, R. E., and Polonsky, I. P., "SNOBOL, A String Manipulation Language", *J. ACM*, v. 11, pp. 21-30, January 1964.

10. Griswold, R. E., *A Guide to the Macro Implementation of SNOBOL4*, Bell Telephone Laboratories, Holmdel, New Jersey, 20 February 1969.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Naval Ship Systems Command (Code 2052) Department of the Navy Washington, D. C. 20360	1
4.	Prof. G. A. Kildall, Code 53Kd Mathematics Department Naval Postgraduate School Monterey, California 93940	1
5.	LTJG Ernest H. Henninger, USN 4965 Eldridge Golden, Colorado 80401	1
6.	Prof. G. L. Barksdale, Jr., Code 53Bv Mathematics Department Naval Postgraduate School Monterey, California 93940	1
7.	Prof. W. S. Brainerd, Code 53Bz Mathematics Department Naval Postgraduate School Monterey, California 93940	1
8.	Professor E. A. Singer, Code 53Sf Mathematics Department Naval Postgraduate School Monterey, California 93940	1
9.	Professor W. M. Waite University of Colorado Department of Electrical Engineering Boulder, Colorado 80302	1
10.	Mr. J. K. M. Moody University Mathematical Laboratory Corn Exchange Street Cambridge, ENGLAND	1

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Naval Postgraduate School Monterey, California 93940	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE
A Study of the Efficiencies in the Mobile Programming System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Master's Thesis; June 1969

5. AUTHOR(S) *(First name, middle initial, last name)*
Ernest Henry Henninger

6. REPORT DATE June 1969	7a. TOTAL NO. OF PAGES 44	7b. NO. OF REFS 10

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

The Mobile Programming System was developed to provide the capability of moving programs from one computing machine to another with a minimum of difficulty. This paper is an initial study of the efficiencies involved in the development of a processor for a programming language via the system. To this end, a language processor was implemented through the system on a particular machine (IBM 360 Mod 67), and comparisons made with the same language processor implemented directly on the same machine. Although the results of this paper are taken from this specific case, they give an indication of the relative efficiencies that could be expected from other processors implemented in a similar way. A significant side benefit of the study is a simplified implementation process for the SNOBOL4 programming language.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

Unclassified
Security Classification

A-31408

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Mobile Programming Systems						
SNOBOL4						
Macro processors						
Machine-independent programming						